Concurrent Probabilistic Planning in the Graphplan Framework

Iain Little and **Sylvie Thiébaux**
National ICT Australia & Computer Sciences Laboratory
The Australian National University
Canberra, ACT 0200, Australia

Abstract

We consider the problem of planning optimally in potentially concurrent probabilistic domains: actions have probabilistic effects and may execute in parallel under certain conditions; we seek a contingency plan that maximises the probability of reaching the goal. The `Graphplan` framework has proven to be highly successful at solving classical planning problems, but has not previously been applied to probabilistic planning in its entirety. We present an extension of the full framework to probabilistic domains that demonstrates a method of efficiently finding optimal contingency plans using a goal regression search. `Paragraph`, the resulting planner, is competitive with the state of the art, producing acyclic or cyclic plans that optionally exploit a problem's potential for concurrency.

Introduction

Planning when the effects of actions can be probabilistically determined—or *probabilistic planning*—differs from classical planning in the need to consider courses of action contingent on the actual outcomes of executed actions. Here, an effective solution is a contingency plan with maximal probability of reaching the goal within a given time horizon.

Most recent research on probabilistic planning is focused on Markov decision process (MDP) models and dynamic programming or state-space search methods (Boutilier, Dean, & Hanks 1999). MDP models typically assume that actions are executed sequentially, and yet real-world problems often require a degree of concurrency. For instance, military operations planning generally involves concurrent tasks for different units, e.g. air and ground forces, each of which may fail with some probability (Aberdeen, Thiébaux, & Zhang 2004); and Mars rover control involves the use of instruments while the robot is moving in a manner that is not completely predictable (Bresina *et al.* 2002).

In this paper, we consider probabilistic planning problems described in the Probabilistic Planning Domain Definition Language (PPDDL) (Younes & Littman 2004), but allowing concurrency. These are equivalent to concurrent Markov decision processes, i.e., MDPs allowing multiple actions per decision step as considered by Mausam and Weld (2004). We consider this problem to be a useful stepping stone in assessing the feasibility of our approach for the more general *probabilistic temporal* planning problem (Little, Aberdeen, & Thiébaux 2005; Mausam & Weld 2005), although it is interesting in its own right. Unfortunately, concurrent MDPs cause an exponential blowup in dynamic programming and state-space search techniques. This leads us to investigate approaches that efficiently deal with concurrency in the deterministic setting, and mitigate the penalty for allowing concurrency by extending them to probabilistic planning.

The `Graphplan` framework (Blum & Furst 1997) is one such approach. Extensions of this framework for probabilistic planning have been developed (Blum & Langford 1999), but either dispense with the techniques that enable concurrency to be efficiently managed, or are unable to produce optimal contingency plans. Specifically, `PGraphplan` finds optimal (non-concurrent) contingency plans via dynamic programming, using information propagated backwards through the planning graph to identify states from which the goal is provably unreachable. This approach takes advantage of neither the state space compression inherent in `Graphplan`'s goal regression search, nor the pruning power of `Graphplan`'s mutex reasoning and nogood learning. `TGraphplan` is a minor extension of the original `Graphplan` algorithm that computes a single path to the goal with a maximal probability of success; replanning could be applied when a plan's execution deviates from this path, but this strategy is not optimal.

Our main contribution is a framework for applying the `Graphplan` algorithm to (concurrent) probabilistic planning, which enables much of the existing research into the `Graphplan` framework for the deterministic setting to be applied to probabilistic planning. `Paragraph` is a planner that implements some of this potential, including: a probabilistic planning graph, powerful mutex reasoning, nogood learning, and a goal regression search. The key to this framework is an efficient method of finding optimal contingency plans using goal regression. This method is fully compatible with the `Graphplan` framework, but is also more generally applicable. Other `Graphplan` optimisations that could be applied to `Paragraph`'s framework include explanation-based learning, invariant synthesis, and constraint-based search optimisations.

The paper starts by defining the probabilistic planning problems that we consider, and the properties of the contingency plans generated by `Paragraph`. We then give an overview of the original `Graphplan` framework, followed by a description of our probabilistic planning graph. We continue with the descriptions of `Paragraph`'s search space and search algorithms for generating both acyclic

and cyclic plans. Finally, we compare the performance of `Paragraph` with state of the art probabilistic planners, and conclude with remarks about related and future work.

Probabilistic Planning Problem

The probabilistic planning problems we consider consist of a finite set of propositions $\mathcal{P}$, an initial state $s_0 \subseteq \mathcal{P}$, a goal $G \subseteq \mathcal{P}$, a finite set of actions $\mathcal{A}$, a finite set of outcomes $\mathcal{O}$, and a time horizon h. Every action a has a set of preconditions $pre(a) \subseteq \mathcal{P}$, and a set of outcomes $out(a) \subseteq \mathcal{O}$. Each outcome o belongs to exactly one action $act(o)$, and has a set of add effects $add(o) \subseteq \mathcal{P}$ and a set of delete effects $del(o) \subseteq \mathcal{P}$. For each action a, there is a probability distribution $\Pr_a(o)$ over the outcomes $out(a)$. The horizon h is finite when acyclic plans are sought, and infinite otherwise.

`Paragraph` adopts a (reward-less) PPDDL-like syntax (Younes & Littman 2004); it directly supports negative preconditions and goals, but as this impacts on the algorithms in only obvious ways we have decided to use the simpler model described above.

Concurrency

A *joint outcome* of a given set of actions $A \subseteq \mathcal{A}$ is a set consisting of exactly one outcome of each action in A. We define the set of possible joint outcomes of A as $Out(A) = \{O \subseteq \mathcal{O} \mid \forall a \in A\ \exists! o \in O \cap out(a)\}$. Every joint outcome O belongs to exactly one action set $Act(O) = \bigcup_{o \in O} act(o)$, and occurs with probability $\Pr(O) = \prod_{o \in O} \Pr_{act(o)}(o)$.

A set of actions A can be concurrently executed in a state $s \subseteq \mathcal{P}$ when: (1) the action preconditions are satisfied in s: $\bigcup_{a \in A} pre(a) \subseteq s$, and (2) the actions have at least one consistent joint outcome. We say that a joint outcome is consistent if: (a) no proposition is both added and deleted, and (b) no proposition that is deleted by one outcome is a precondition of another's action. That is, $\exists O \in Out(A)$ such that:

(a) $\bigcup_{o \in O} add(o) \cap \bigcup_{o \in O} del(o) = \emptyset$, and

(b) $\forall o \in O\ \ del(o) \cap \bigcup_{a \in A \setminus \{act(o)\}} pre(a) = \emptyset$.

The result of joint outcome O occurring in state s is $res(s, O) = (s \setminus \bigcup_{o \in O} del(o)) \cup \bigcup_{o \in O} add(o)$ if O is consistent and $\perp$ otherwise. It would be reasonable to insist that all joint outcomes be consistent for a set of actions to be executed, but as we explain in (Little, Aberdeen, & Thiébaux 2005), it can be useful to find solutions that only satisfy the weaker condition. Optimising planners tend to find solutions where a conflicting set of outcomes is as unlikely as possible, but allowing potential conflicts can sometimes be necessary for a solution to be possible at all.

It can often be desirable to restrict the degree of concurrency that is allowed, e.g., to reduce the size of the search space. This applies particularly to domains that do not admit concurrent solutions. We consider three different models for managing concurrency (the second of which only applies to goal-directed planners):

Unrestricted: actions are permitted to execute concurrently as defined above, without any additional restrictions.

Restricted: a pair of actions are additionally prevented from executing concurrently when a pair of 'preferred outcomes' both add or both delete the same proposition. By 'preferred outcomes', we mean a pair of outcomes that the search algorithm has determined to be desirable for achieving the goal, and not a chance contingency that incidently needs to be considered to make the plan complete. This model is intended to allow the planner to produce plans that attempt to achieve each subgoal in one way at a time, but to not prohibit chance occurrences that just happen to achieve a subgoal in multiple ways. In many domains, solving a problem using this concurrency model will produce the same solution as the unrestricted model.

Non-concurrent: actions are not permitted to execute concurrently under any circumstance.

`Paragraph` implements the non-concurrent and restricted concurrency models. Its algorithms could easily be applied to the unrestricted concurrency model, although there is likely to be a significant performance overhead.

Plans

In order to encompass both acyclic and cyclic solutions, we define a plan as a deterministic finite state automaton $\langle Q, q_0, M, \delta, F \rangle$, where Q is the finite set of plan states or "steps", $q_0 \in Q$ is the initial plan step, $M : Q \rightarrow 2^{\mathcal{A}}$ is the labelling function that prescribes the set of actions to be concurrently executed at a given plan step, $\delta : Q \times 2^{\mathcal{O}} \rightarrow Q$ is the plan's partial transition function such that $\delta(q, O)$ is defined iff $O \in Out(M(q))$, and $F \subseteq Q$ is the set of final plan steps. All final plan steps—and no others—are labelled with the empty action set.

We define the function $L : Q \rightarrow 2^{\mathcal{P}} \cup \{\perp\}$ that intuitively maps each plan step to the world's state when that step is executed. A plan is valid iff the length of longest path from q_0 to a final step in F does not exceed the horizon h, and there exists an L such that: $L(q_0) = s_0$, $L(q) \neq \perp$ for every non-final plan step $q \in Q \setminus F$; and for every plan step $q \in Q$, the set of actions $M(q)$ can be concurrently executed in world state $L(q)$ and $L(\delta(q, O)) = res(L(q), O)$ for every joint outcome $O \in Out(M(q))$.

The cost $\mathcal{C}(q_0)$ of a plan is the expected probability of failing to reach the goal[1] and is defined recursively as follows:

$$\mathcal{C}(q) = \begin{cases} 0 & q \in F \text{ and } L(q) \supseteq G \\ 1 & q \in F \text{ and } L(q) \not\supseteq G \\ \displaystyle\sum_{O \in Out(M(q))} \Pr(O) \times \mathcal{C}(\delta(q, O)) & \text{otherwise} \end{cases}$$

A plan is *optimal* when its cost is minimal for the given solution space. We say that a plan is *non-redundant* when it contains no actions that could be omitted without affecting the plan's validity or cost.[2] There always exists a plan that is both non-redundant and optimal (e.g. any optimal plan with all redundant actions removed).

[1] Although we do not explore this angle, `Paragraph` could be used to generate optimal plans for (concurrent) stochastic shortest path problems (Bonet & Geffner 2003; Mausam & Weld 2004), given the cost of individual actions (and assuming additive costs for action sets).

[2] Testing whether an action can be omitted could be done by setting the precondition of this action to true and the effects of all its outcomes to $\emptyset$, leaving the structure of the plan intact.

A (valid) *trajectory* $q_0 \stackrel{O_0}{\rightarrow} q_1 \ldots \stackrel{O_{n-1}}{\rightarrow} q_n$ with $n \leq h$ is a path in the plan from the initial step to a final step at which the goal is reached, i.e., it satisfies $L(q_0) = s_0$, $L(q_n) \supseteq G$, and for $i = 0 \ldots n-1$, the actions in $M(q_i)$ can be concurrently executed in $L(q_i)$ and $L(q_{i+1}) = res(L(q_i), O_i) \neq \perp$. We say that a trajectory is non-redundant when it contains no action that could be omitted without affecting the validity of that trajectory.

A *composite contingency plan* is a contingency plan where for each step there exists a plan trajectory that is contributed to by all of the step's actions: none of the actions could be omitted without affecting the trajectory's validity. Intuitively, a composite contingency plan is composed of an interleaving of non-redundant trajectories. For any given problem, an optimal composite contingency plan will always have the same cost as an optimal contingency plan as long as the solution size is not restricted; a naive mapping from contingency to composite contingency plans would split plan steps containing multiple actions into smaller steps until the composite contingency property is satisfied.

Given a problem and concurrency model, `Paragraph` will find a (non-redundant) optimal solution from the space of valid composite contingency plans.

Graphplan

The `Graphplan` framework defines an efficient method of solving classical planning problems. The original `Graphplan` planner (Blum & Furst 1997) could find potentially concurrent solutions to STRIPS domains. Its framework has since been extended in a variety of ways. Examples include *negative preconditions*, *conditional effects* (Kambhampati, Parker, & Lambrecht 1997; Koehler *et al.* 1997), and *durative actions* (Smith & Weld 1999).

`Graphplan`'s most notable contribution is the concept of a *planning graph*, which is a polynomial size data structure based on a relaxation of the reachability problem; it yields a necessary but insufficient condition for the reachability of a set of propositions. A planning graph consists of alternate levels of proposition and action nodes; each level represents the union of what might be reachable by a given time step. The initial level consists of nodes for each of the initial propositions, and the first action level consists of nodes for each of the initially executable actions. The second proposition level consists of the add effects of the first action level. Successive levels are generated by inference according to action preconditions and effects. Special actions are used to represent the persistence of a proposition from one time step to the next. Such *persistence actions* have a single proposition as both a precondition and add effect, and for algorithmic purposes are included in the set of actions.

Planning graphs can be made a more accurate representation of action and proposition reachability by the use of binary mutual exclusion relationships; referred to as *mutexes*. For any given level, we say that a pair of nodes are mutex when they might be individually reachable, but cannot both be achieved concurrently. There is a set of rules that are used to determine some of the situations when a pair of propositions or actions are mutex. These rules are based on reasoning about impossible situations and resource contention.

The `Graphplan` algorithm starts by creating a planning graph for the given problem. The graph is incrementally expanded until the final level includes nodes for all goal propositions and there are no binary mutex relationships between them, at which point the goal might be reachable. A backward search through the structure of the planning graph is then performed to extract a potentially concurrent solution of a length equal to the number of action levels. If no such solution exists, then the planning graph is expanded again and another search is performed. This continues until either a solution is found or the termination condition is met. When the content of the final level does not change between successive expansions, the graph is said to have *levelled off*. If the problem does not have a solution, this can then be proven after a finite number of additional expansions (Blum & Furst 1997); it can be proven immediately if the graph levels off before all goal propositions are present, or while there are still mutexes between goal propositions.

Mutexes play a direct role in speeding up the solution extraction search. When a branch is encountered that violates a known mutex relationship, then search is able to backtrack and consider the next possibility. New, potentially larger exclusion conditions (*nogoods*) can also be 'learnt' when branches are closed due to the exhaustion of possibilities. This form of learning is a powerful technique, and benefits both current and future attempts to extract a solution.

`Paragraph` extends this framework for finding optimal (composite) contingency plans in the probabilistic setting. Importantly, `Paragraph` does this while retaining the framework's distinctive features: the planning graph, a goal regression search for solution extraction, and the use of mutual exclusion information for search space pruning.

Probabilistic Planning Graph

A structural change to the standard planning graph is required to accurately represent actions with probabilistic effects (Blum & Langford 1999; Little, Aberdeen, & Thiébaux 2005). `Paragraph` does this through the addition of a node for each of an action's possible outcomes, so that there are three different types of nodes in the graph: proposition, action, and outcome. Action nodes are then linked to their respective outcome nodes, and edges representing effects link outcome nodes to proposition nodes. Each persistence action has a single outcome with a single add effect. We refer to a persistence action's outcome as a *persistence outcome*.

The mutex rules that `Paragraph` uses are similar to the standard ones. The differences all relate to the introduction of outcome nodes. We only consider mutex relationships between nodes of the same level. This gives us three different types of mutexes: action, outcome and proposition. The conditions for recognising action node mutexes are:

1. The actions have *competing needs*. This occurs when a pair of respective precondition nodes are mutex.
2. All pairs of respective outcome nodes are permanently mutex. We refer to this condition as *contention*.[3]

[3] As mentioned previously, it could be reasonable to use a stronger model that prohibits the possibility of any inconsistencies between outcomes, in which case this condition should be changed to consider a pair of actions mutex if *any* of their outcomes are.

The graph alternates layers of propositions, action, and outcome nodes $P_0, A_1, O_1, P_1, A_2, O_2, \ldots, A_k, O_k, P_k$ and records mutex pairs μA_i, μO_i, μP_i at each layer and permanent mutexes μO_∞, such that:

1. $P_0 = s_0$,
2. $\mu P_0 = \emptyset$,
3. $A_{i+1} = \{a \in A \mid pre(a) \subseteq P_i \ \wedge \ \forall \{p, p'\} \in \mu P_i \ \{p, p'\} \not\subseteq pre(a)\}$,
4. $\mu A_{i+1} = \{\{a, a'\} \subseteq A_{i+1} \mid Out(\{a, a'\}) \subseteq \mu O_\infty$
   $\vee \exists \{p, p'\} \in \mu P_i \ p \in pre(a) \wedge p' \in pre(a')\}$,
5. $O_{i+1} = \bigcup_{a \in A_{i+1}} out(a)$,
6. $\mu O_{i+1} = \{\{o, o'\} \subseteq O_{i+1} \mid Act(\{o, o'\}) \in \mu A_{i+1}\} \ \cup \ \mu O_\infty$
7. $P_{i+1} = \bigcup_{o \in O_{i+1}} add(o)$,
8. $\mu P_{i+1} = \{\{p, p'\} \subseteq P_{i+1} \mid \forall o, o' \in O_{i+1} \ (p \in add(o)$
   $\wedge \ p' \in add(o')) \Rightarrow \{o, o'\} \in \mu O_{i+1}\}$, and
9. $\mu O_\infty = \{\{o, o'\} \subseteq \mathcal{O} \mid act(o) = act(o')$
   $\vee \ del(o) \cap (pre(act(o')) \cup add(o')) \neq \emptyset\}$.

Figure 1: The definition of `Paragraph`'s planning graph.

In the non-concurrent model, all pairs of non-persistence actions are always mutex (giving us a *serial* planning graph). The conditions for permanent outcome node mutexes are:

1. The outcomes have *inconsistent effects*: when a proposition added by one outcome is deleted by the other.
2. There is *interference* between the effects of one outcome and the preconditions of the other's action. This is when one outcome deletes a precondition of the other's action.
3. The outcomes are *exclusive*. This occurs when both outcomes belong to the same action.

These conditions are independent of the level in the planning graph, and can be precomputed before generating the graph. A pair of outcome nodes are non-permanently mutex when the respective action nodes are mutex. A pair of proposition nodes are mutex when they have *inconsistent support*. This occurs when all pairs of outcomes that support the respective propositions are themselves mutex. A formal definition of `Paragraph`'s planning graph is given in Figure 1.

The implementation of `Paragraph`'s planning graph is loosely based on the compact representation described by Long and Fox (1999). We have made an effort to minimise the amount of computation required to determine the mutex relationships by taking advantage of the planning graph's monotonicity properties (Smith & Weld 1999) and reasoning about how the mutexes can change from one level to another.

Search Space

The solution extraction step of the `Graphplan` algorithm relies on a backward search through the structure of the planning graph. In classical planning, the goal is to find a subset of action nodes for each level such that the respective sequence of action sets constitutes a valid trajectory. The search starts from the final level of the graph, and attempts to extend partial trajectories one level at a time until a solution is found.

Solution extraction can also be described as a regression search over the space of goal sets. Each extension of a partial plan requires a set of actions to be selected to support a set of goal propositions. The preconditions of the actions selected for one extension make up the goal set that must be supported by the next. In classical planning, a problem is considered solved when a trajectory from the initial conditions to the goal is found. In the probabilistic setting a single trajectory is generally not considered sufficient, as we want plans to include the contingencies needed to maximise their probability of executing successfully.

`Paragraph` uses this type of goal-regression search with an explicit representation of the expanded search space. This search is applied exhaustively, to find all trajectories that can be found when admissible pruning is applied. An optimal contingency plan is formed by linking these trajectories together. This requires some additional computation, and involves using forward simulation through the search space to compute the possible world states at reachable search nodes. The search space is defined as:

1. a set of nodes $\mathcal{N}$,
2. an initial node n_0,
3. a set of goal nodes $\mathcal{G} \subseteq \mathcal{N}$,
4. a state function $W : \mathcal{N} \rightarrow 2^{2^{\mathcal{P}}}$,
5. a cost function $C : \mathcal{N} \times 2^{\mathcal{P}} \rightarrow [0, 1]$,
6. a selection function $S : \mathcal{N} \rightarrow 2^{2^{\mathcal{A}}}$,
7. a successor function $T : \mathcal{N} \times 2^{\mathcal{O}} \rightarrow 2^{\mathcal{N}}$, and
8. a conditional successor function $J : \mathcal{N} \times 2^{\mathcal{O}} \times 2^{\mathcal{P}} \rightarrow 2^{\mathcal{N}}$.

The exact definition of a node depends on whether the search is cyclic or acyclic. For instance, an acyclic search node is defined as: a time step t_n, a goal set $G_n \in \mathcal{P}$, and a label $l_n \in \{\texttt{success}, \texttt{failure}, \texttt{unknown}\}$. A node's time is the index of an associated proposition level; a node of time t can only have predecessors of time $t - 1$. The initial node n_0 has a time of 0. The goal sets of the initial and goal nodes are derived from the problem, giving $G_{n_0} = s_0$ and $\forall n \in \mathcal{G} \ G_n = G$. Each goal node $g \in \mathcal{G}$ is created for a particular iteration of the solution extraction step; its time is that of the current final graph level. A node is uniquely identified by its time and goal set; there can be at most one node with a particular goal set for any given time. These constraints make the search acyclic. A node's label is used by the search to keep track of whether the node is reachable or provably unreachable—in the forwards direction—from the initial state, or that we do not have enough information yet to determine the node's status. The cyclic search's node is defined along with the algorithm's description.

The selection function maps a node to its set of selectable action sets. We use the term successor in the sense of taking a step towards the goal. The successor function maps a node and a joint outcome to a set of potential successor nodes. That individual joint outcomes can have more than one successor is an artifact of the goal regression search, as a joint outcome is able to support multiple goal sets.

The state function maps a node to a set of possible world states: the world states that are possible when the node is reached during a simulation of the known trajectories. The conditional successor function maps a node, a joint outcome, and a world state to a set of potential successor nodes. It is used to link disparate trajectories together when some successors only apply to a subset of a node's possible world states. We define the union of all applicable successor nodes as $N(n, O, s_n) = T(n, O) \cup J(n, O, s_n)$.

The cost function maps a search node and a world state to a probability of failure. It is a consequence of the conditional

successors that a node does not have a unique cost. The cost function initially maps every node and state to a cost of 1. As the search space is expanded, the costs are updated using the following formula:

$$C(n, s_n) := \begin{cases} 0 & n \in \mathcal{G} \\ \min\limits_{A \in S(n)} C_A(n, s_n, A) & \text{otherwise} \end{cases}$$

where

$$C_A(n, s_n, A) := \sum_{O \in Out(A)} \Pr(O) \times \min_{n' \in N(n,O,s_n)} C(n', res(s_n, O)).$$

The backward search algorithms that `Paragraph` uses with this search space both determine a node's predecessors in the same way; using the sets of outcomes that consistently support the node's goal set. We say that a set of non-mutex outcomes O_P consistently supports a set of propositions P when all outcomes add at least one proposition in P, no outcomes delete any propositions in P, and every proposition in P is supported by at least one of the outcomes. So that the regression corresponds to the `Graphplan` search, we only consider outcome sets where every outcome has a corresponding node in one of the planning graph's levels.[4] We also disregard outcome sets that either consist entirely of persistence actions or include persistence actions that are unnecessary to support all goal propositions. Using the restricted concurrency model to prohibit multiple support for individual subgoals adds the rule $\bigcap_{o \in O_P} add(o) = \emptyset$, and O_P is required to contain exactly one non-persistence action when concurrency is disallowed entirely.

Each resulting consistent set of outcomes O_P leads to a predecessor node n_{O_P}. The goal set of each predecessor node is determined by the preconditions of the respective outcome set's actions $\bigcup_{a \in Act(O_P)} pre(a)$. Persistence actions are not included in the selection function's action sets, and the corresponding outcomes are not included in successor function outcome sets. It is possible for a node to be a predecessor of another in multiple ways.

Acyclic Search

`Paragraph`'s search uses goal regression to exhaustively search for trajectories and links them together to form an optimal contingency plan. As observed by Blum and Langford (1999), the difficulty with combining probabilistic planning with `Graphplan`-style regression is in correctly and efficiently combining the trajectories. Our solution to this problem is a key part of our contribution.

`Paragraph`'s acyclic search preserves the structural framework of the `Graphplan` algorithm: a planning graph is incrementally expanded until it might be possible to reach the goal, a backward search is then alternated with future graph expansions, and a solution is ultimately extracted. The difference is that `Paragraph` is looking for contingency plans: it does not (normally) terminate when a single trajectory is found, and needs to do some extra computation to be sure to solve all of the contingencies.

[4]The exact level depends on whether the search is cyclic or acyclic and is detailed in the respective sections.

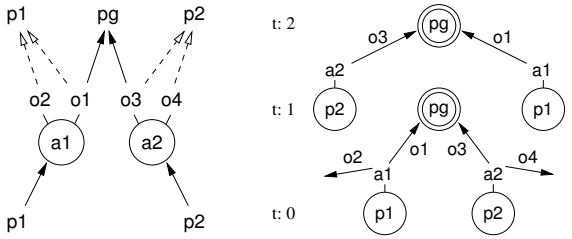

Figure 2: An action-outcome-proposition dependency graph and search space for an example problem.

Nogoods

`Paragraph` keeps track of nogood tuples for each time step. These nogoods are sets of propositions that cannot be achieved by the given time, and initially consist of the planning graph's proposition mutexes. New nogoods are learnt from the goal sets of nodes labelled a `failure` due to an exhaustion of possibilities. The search space is pruned whenever a node can be proved not to be part of a successful trajectory: when for a given node n of time t, there exists a nogood P of time t' such that $P \subseteq G_n$ and $t \leq t'$. The reasoning over different times is sound because if P cannot be part of a trajectory at a given time, then it also cannot be part of a trajectory at any earlier, more constrained time.

Nogood pruning is implemented using a data structure that we refer to as a *subset memoizer*. This data structure is responsible for maintaining a set of stored—or *memoized*—tuples. It is based on the trie data structure, and enables us to efficiently determine when one of the stored tuples is a subset of a given set. `Paragraph`'s subset memoizer is similar to that described by Koehler *et al.* (Koehler *et al.* 1997). The most important difference is that `Paragraph`'s memoizer for nogoods recognises that a nogood which is valid for a given time also applies to all earlier times. This is achieved by labelling the trie's nodes with expiry times.

Joining Trajectories

`Paragraph`'s search is able to find a trajectory for each way of achieving the goal from the initial state. Sometimes these trajectories will 'naturally' join together during the regression, which happens when nodes share a predecessor through different joint outcomes of the same action set.

Unfortunately, the natural joins are not sufficient to find all contingencies. Consider the problem shown in Figure 2, which we define as:[5] the propositions $p1$, $p2$ and pg; $s_0 = \{p1, p2\}$; $G = \{pg\}$; the actions $a1$ and $a2$; and the outcomes $o1$ to $o4$. $a1$ has precondition $p1$ and outcomes $\{o1, o2\}$; $a2$ has precondition $p2$ and outcomes $\{o3, o4\}$. Both actions always delete their precondition; $o1$ and $o3$ both add pg. To simplify the example, we prohibit $a1$ and $a2$ from executing concurrently. The optimal plan for this example is to execute one of the actions; if the first action does not achieve the goal, then the other action is executed.

[5]This problem was used by Blum and Langford (1999) to illustrate the difficulty of using goal-regression for probabilistic planning, and to explain their preference of a forward search in `PGraphplan`.

The backward search will correctly recognise that executing $a1$–$o1$ or $a2$–$o3$ will achieve the goal, but it fails to realise that $a1$–$o2, a2$–$o3$ and $a2$–$o4, a1$–$o1$ are also valid trajectories. The longer trajectories are not discovered because they contain a 'redundant' first step; there is no way of relating the effect of $o2$ and the precondition of $a2$, or the effect of $o4$ with the precondition of $a1$. While these undiscovered trajectories are not the most desirable execution sequences, they are necessary for an optimal contingency plan. In classical planning, it is actually a good thing that trajectories with this type of redundancy cannot be discovered, as redundant steps only hinder the search for a single shortest trajectory. Identifying the missing trajectories requires some additional computation beyond the goal regression search. When the distinction is necessary, we refer to trajectories that can be found using unadorned goal regression as *natural trajectories*. In `Paragraph`, the set of natural trajectories is equivalent the set of non-redundant trajectories.

One way of finding all necessary trajectories would be to allow the backward search to consider all non-mutex outcome sets when computing a node's predecessors, as opposed to only considering outcomes that support the node's goal propositions. We consider it likely that this solution would be prohibitively expensive; certainly it compromises a lot of the pruning power inherent in goal regression.

Non-concurrent Trajectories The solution that we have developed is based on constructing all non-redundant contingency plans by linking together the trajectories that goal regression is able to find. This is sufficient to find an optimal solution, as there always exists at least one non-redundant optimal plan. Let us first consider the non-concurrent case. The key observation—that makes this approach feasible—is that all undiscovered trajectories required for a non-redundant contingency plan consist entirely of subsets of the natural trajectories. We now give a proof of this.

Proof. Assume that some non-redundant plan contains exactly one non-terminal node n that is not a member of any natural trajectory.[6] Let A be n's action set in the plan, giving n a goal set of $G_n = \bigcup_{a \in A} pre(a)$. If there exists a set of outcomes $O \in Out(A)$ and persistence outcomes O' such that $O_P = O \cup O'$ consistently supports a successor of n, then either the successor is not a natural trajectory node, or n would be found using goal regression and either be unreachable from the initial state or a member of a natural trajectory. All of these cases violate an assumption, and so there is no O_P that consistently supports a successor. Now, as the plan is non-redundant, there must be at least one successor that is supported by at least one (non-persistence) outcome $o \in O$ for some $O \in Out(A)$; A is irrelevant to the success of the plan if no joint outcome O adds at least one successor's goal $p \notin G_n$. But the plan is non-concurrent, so such an $O = \{o\}$ would mean that there exists an O_P that does consistently support a successor, and we have a contradiction. □

This proof can be generalised to the case where there are multiple nodes not of a natural trajectory by realising that there must always be some node that has only natural and terminal successors, and applying induction. We do not need to consider terminal nodes in the proof as they must either be a failure or a goal node. A node of the former case is irrelevant, and in the later case a goal node that is part of any trajectory is necessarily a member of some natural trajectory. This result means that it is sufficient to link together pairs of known trajectories to construct an optimal non-concurrent solution; the contingency trajectories that consist of more than two natural trajectory subsets can be formed with successive joins.

[6] Every plan trajectory corresponds to a search space trajectory.

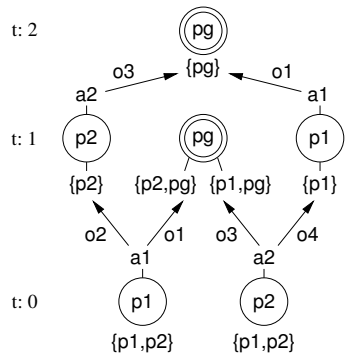

Figure 3: The previous example's search space after joining trajectories using world state information. Each possible world state is shown below its respective search node.

Concurrent Trajectories The previous proof also gives us some insight into the concurrent case, as the proof itself relies on a lack of concurrency. In fact, it is possible to construct a problem where an optimal solution that fully exploits concurrency must contain a node that is not a member of any natural trajectory. This happens when steps from disparate trajectories can be executed in parallel to reduce the size of the solution, and requires a set of non-persistence outcomes that the goal regression search will never consider. So in addition to linking together subsets of the natural trajectories, fully exploiting concurrency may require that some nodes be merged. It would seem that merging nodes could be made feasible, but we have not attempted to do this. In practice, we can produce useful concurrent solutions even if we ignore this potential source of unexploited concurrency.

To clarify, `Paragraph` can produce a given non-redundant concurrent solution iff this solution requires only search nodes that are a member of some natural trajectory. That is, `Paragraph` can find all composite contingency solutions to concurrent planning problems, at least one of which will be an optimal composite contingency plan.

Join Operation `Paragraph` combines pairs of trajectories by making a node in one trajectory a conditional successor of a node in the other. This can be done when a possible world state of the earlier node has a resulting world state that subsumes the goal set of the later node. Specifically, we define the join operation $\mathcal{J}(n, O, s, n')$, where $n, n' \in \mathcal{N}$, $O \in Out(A)$ for some $A \in S(n)$, $s \in W(n)$ is a possible world state for n, $res(s, O) \supseteq G(n')$, and n is for the time step immediately preceding that of n'. This operation makes n' a successor of n that is conditional on s; the conditional

successor function is updated so that $n' \in J(n, O, s)$. For a given join, we say that n is the *source* node, and that n' is the *target* successor node. The result of applying the join operation to our previous example is shown in Figure 3, where all of the trajectories needed for an optimal contingency solution are now part of the search space.

The world states needed for this operation are computed by simulating the possible search space trajectories in the forward direction. This simulation starts with the initial state at the `success` nodes of time 0, and recursively visits all applicable successor nodes. Every `success` node will be visited at least once; any node that is not a `success` is not part of a successful trajectory, and cannot be reached by a forward traversal of the search space.

A potential join can only be detected when the target node has been visited by the backward search, and the applicable world state of the source node has also been computed. Consequently, `Paragraph` looks for potential trajectory joins whenever either of these events occur: when a search node is visited for the first time, and when a world state is computed for a `success` node. To facilitate this detection, all `success` nodes are indexed according to their goal set, and all node/joint outcome pairs are indexed according to their possible join target world states.

Join detection for a newly computed world state s of node n looks up each `success` node n' where $res(s, O) \supseteq G_{n'}$ for some joint outcome $O \in Out(A)$ for each selectable action set $A \in S(n)$. This lookup is implemented using another subset memoizer, which is the data structure used to index `success` nodes according to their goal sets. In contrast to nogood memoization, these nodes are indexed only for their own specific time: the trie nodes are not labelled with expiry times, and a separate memoizer is maintained for each of the planning graph's proposition levels. The subset memoization test is adapted into a procedure that collects each detectable join target n'. Join detection when the search first visits a node n' is similar: potential join source nodes are looked up using the subset memoizer data structure, where node/joint outcome pairs are indexed according to target world states for the respective target node time. There is a slight complication due to each target state s' being a superset of the node's goal set: we really need to do a superset lookup for $G_{n'}$, which is not what the subset memoizer is optimised for. We transform the superset test into a subset test by taking the complement of both the goal set and candidate target states. The memoizer then finds each world state complement that is subsumed by the node's goal set complement, where $\mathcal{P} \setminus s' \subseteq \mathcal{P} \setminus G_{n'}$.

Backward Search

A depth-first search is used by `Paragraph` for goal regression. We chose this in preference of a breadth-first search to increase the chance that nogood sets are found early enough to be effective at pruning the current iteration of the backward search. The recursive algorithm for the backward search is given in Figure 4. It takes a node n as input. If the node has been previously solved, then there is nothing to do. A node is a `failure` if its goal set subsumes a nogood. If neither of the previous conditions apply, then the search is recursively called on each of n's predecessors. A node n is a `success` if any of its predecessors are, or if it satisfies the initial conditions $s_0 \supseteq G_n$; if n is not a `success` then it is a `failure`. Not all trajectories include a time 0 node, although they will include the same sequence of actions and outcomes as a trajectory that does. We compute these 'duplicate' trajectories because the optimal contingency plan needs to be able to combine structurally identical trajectories at different time offsets, and there would be a (limited) potential for cycles if we were to allow a join's source node to have a later time than its target.

BACKWARD-SEARCH(n): Expand node n in a recursive depth-first search.

1. If $l_n \neq$ `unknown` then return.
2. If G_n is a nogood applicable to time t where $t_n \leq t$, then set $l_n :=$ `failure` and return.
3. Determine the predecessors of n; for each computed predecessor n':
   (a) recursively call BACKWARD-SEARCH(n'), and
   (b) if $l_{n'} =$ `success` then $l_n :=$ `success`.
4. If $s_0 \supseteq G_n$ then $l_n :=$ `success`; if $l_n \neq$ `success` then $l_n :=$ `failure`.

Figure 4: The acyclic goal regression search algorithm.

`Paragraph` does not use any heuristics to guide the backward search, beyond the planning graph. Since this search must be exhaustive to guarantee that all contingencies are found, it is not obvious what the best way of incorporating search heuristics into `Paragraph`'s framework would be. Perhaps there is some way of guiding the search to find effective nogood sets quickly, or some test for determining when a trajectory is unlikely to have much impact on the final solution. At present, `Paragraph` relies on nogood pruning and the compressed search space for efficiency.

The forward simulation for computing the possible world states is performed after each iteration of the backward search. `Paragraph` keeps track of the search iteration in which each node is first encountered so that this simulation can avoid unnecessarily recomputing world states. This opportunity is also used to update the cost function: node/state cost values are propagated in the backwards direction using the cost update formula.

Each node/state pair corresponds to a potential step in a plan. The search space's `success` nodes represents the union of all solutions that have been found. `Paragraph` extracts an optimal solution for the current horizon through a greedy simulation in the forward direction. The construction of a solution automaton starts by making the initial plan step q_0 correspond to a minimal-cost node/world state pair with time 0. Each time a new step q is added to the automaton, $M(q)$ is set to the optimal action set for the node/state pair (n, s) corresponding to q, that is to the set of actions A maximising $C_A(n, s, A)$. For each $O \in Out(A)$, another step q' is added for the optimal successor in $N(n, O, s)$ and its world state $res(s, O)$ (unless this step is already in the plan); a new transition $\delta(q, O, q')$ is also added. If $N(n, O, s) = \emptyset$, then q' is added to the set of final states F. In principle, a solution could be extracted from the search space at any time. `Paragraph` computes the world states and updates the cost values used to determine which solution to extract en masse after each search iteration, but this could be done incrementally during the search.

Algorithm

Now that we have built up the necessary foundation, we give a detailed description of Paragraph's acyclic search algorithm. The first step is to generate a planning graph from the problem specification. This graph is expanded until all goal propositions are present and not mutex with each other, or until the graph levels off to prove that no solution exists. Assuming the former case, a depth-first goal regression search is performed from a goal node for the graph's final level. This search exhaustively finds all natural trajectories from the initial conditions to the goal. Once this search has completed, the possible world states for each trajectory node are computed by forward-propagation from time 0, and the node/state costs are updated by backward-propagation from the goal node. Potential trajectory joins are detected each time a new node is encountered during the backward search, and each time a new world state is computed during the forward state propagation. Unless a termination condition has been met, the planning graph is then expanded by a single level, and the backward search is performed from a new goal node that is added to the existing search space. This alternation between backward search, state simulation, cost propagation, and graph expansion continues until a termination condition is met. An optimal contingency plan is then extracted from the search space by traversing the space in the forward direction using a greedy selection policy.

It is interesting to note that Paragraph alternates the use of forward and backward reasoning, with: forward planning graph expansion, backward goal regression, forward state simulation, backward cost propagation, and forward solution extraction.

Deciding when to terminate a search is not always trivial. Paragraph generally terminates when: a solution of cost 0 is found, the planning graph proves that no solution is possible by levelling off before the backward search can start, or a finite horizon is exceeded. These conditions work well enough as long as a reasonable finite horizon can be specified. However, this might not always be possible. A real time limit is likely to be needed in such situations, where the best discovered solution is extracted from the search space when this limit is reached. In some situations other conditions might be appropriate, such as terminating when the optimal cost exceeds a probability threshold, or when the change in the optimal cost has been small enough over a given number of horizon extensions.

Cyclic Search

Classical planning problems have the property that the shortest solution to a problem will not visit any given world state more than once. This is no longer true for probabilistic planning, as previously visited states can unintentionally be returned to by chance. Because of this, it is common for probabilistic planners to allow for cyclic solutions. We now describe Paragraph's method of producing such solutions. This method departs further from the Graphplan algorithm than the acyclic search does: fundamental to the Graphplan algorithm is a notion of time, which we dispense with for Paragraph's cyclic search.

The cyclic search does not preserve Graphplan's alternation between graph expansion and backward search: the planning graph is expanded until it levels off, and only then is the backward search performed. As there is no notion of time, the backward search is constrained only by the information represented in the final level of the levelled-off planning graph. A search node is defined as: a goal set $G_n \in \mathcal{P}$, and a label $l_n \in \{\texttt{success}, \texttt{failure}, \texttt{unknown}\}$. A search node n is not associated with a time at all, and is uniquely identified by its goal set G_n. This divorce with time make it possible for solutions to contain cycles, and ensures that the search space is finite.

Paragraph uses either a depth-first or iterative deepening algorithm for its cyclic search. In both cases, the search uses the outcomes supporting the planning graph's final level of propositions when determining a search node's predecessors. The same principal is applied to nogood pruning: only the mutexes in the final level of the planning graph—those that are independent of time—can be safely used. An important consequence of only using universally applicable nogoods is that any new nogoods learnt from failure nodes are also universal. Neither search strategy is clearly superior. The depth-first search is usually preferable when searching the entire search space, as it is more likely to learn useful nogoods. A consequence of this is that there is no predictable order in which the trajectories are discovered. In contrast, the iterative deepening search will find the shortest trajectories first, which can be advantageous when only a subset of the search space might be explored. Both search algorithms propagate node labels backwards so that a node is a success when any of its predecessors are, and a failure when they all are.

Most other aspects of the cyclic search are either the same as acyclic case, or only need minor amendment: trajectories are joined in the same way, although the subset memoizers used for join detection only need to be maintained for the final graph (proposition) level; forward state simulation only needs to ensure termination by keeping track of which node/state pairs have been encountered previously; and solution extraction does not need to change. Cost propagation does differ significantly from the acyclic case in order to accommodate cycles. As in many MDP algorithms, the cost function is updated by iterating Bellman backups until convergence within an ϵ.

The cyclic search's termination conditions are simpler than for the acyclic search. Except in the special cases where plan optimality or the lack of a solution can be proved early, the search terminates when the search space has been exhausted; there is no finite horizon. Other termination conditions can still be useful, but are not necessary.

The cyclic search has several clear advantages over the acyclic one. First and foremost is the potential for cyclic solutions. The cyclic search can find optimal solutions to infinite horizon problems in general, while the acyclic search cannot. In terms of performance, the cyclic search also has the advantage of a smaller search space and less redundancy; the acyclic search repeatedly searches over some of the same structure each time the search horizon is extended, while the cyclic search need consider at most one node for each goal set. And the cyclic search does not require any arbitrary restrictions to guarantee termination. The acyclic search does retain some important advantages. In particular, the restric-

tions on its search space allow much more specific mutex conditions to be applied to individual search nodes. For many problems, this additional pruning power more than makes up for the redundancy in the search space. And sometimes there are real limits to how long a plan has to achieve the goal, in which case optimality requires a finite horizon.

Experimental Results

We benchmark Paragraph using a selection of concurrent and non-concurrent problems; we compare our results with mGPT (Bonet & Geffner 2005) and Prottle (Little, Aberdeen, & Thiébaux 2005). Prottle implements the unrestricted model of concurrency, and in some instances can exploit concurrency to a greater degree than Paragraph. mGPT only finds non-concurrent solutions. Prottle and Paragraph are implemented in Common Lisp, and were both compiled using CMUCL version 19c. mGPT is implemented in C++, and was compiled using gcc 2.95. All experiments were performed on a machine with a 3.2 GHz Intel processor and 2 GB of RAM.

We consider five different problems: g-tire, maze, machineshop, zeno-travel and teleport.[7] Both the g-tire and zeno-travel problems are from the probabilistic track of the *2004 International Planning Competition* (IPC-04); maze and teleport are timeless versions of problems used to benchmark Prottle in (Little, Aberdeen, & Thiébaux 2005); machineshop is a problem from (Mausam & Weld 2005). We present results for the following planner configurations: mGPT with LRTDP (Bonet & Geffner 2003), $\epsilon = 0.001$ and the min-min state relaxation heuristic (M-GPT); Prottle with $\epsilon = 0$ and its cost-based planning graph heuristic (PRTTL); Paragraph with its acyclic search using either the restricted concurrency model (CA-PG) or no concurrency (NA-PG); cyclic Paragraph with restricted concurrency and either a depth-first (CCD-PG) or iterative deepening (CCI-PG) search, and cyclic Paragraph without concurrency (NCD-PG,NCI-PG). Some of the heuristics that mGPT supports are stronger than the min-min state relaxation; we used it because the stronger heuristics resulted in suboptimal solutions for some problems. We use the max-propagation variant of Prottle's heuristic; it is admissible under conditions similar to those of the restricted concurrency model, but slightly less permissive. The results are shown in Figure 5. The solution costs for both Paragraph and Prottle were computed analytically; the costs for mGPT were computed empirically through simulation. All times are given in seconds.

The objective of the g-tire problem is to move a vehicle from one location to another, where each time the vehicle moves there is a chance of it getting a flat tire. There are spare tires at some of the locations, and these can be used to replace flat tires. This problem is not concurrent. The results compare Prottle to Paragraph's acyclic search; Paragraph is faster for the earlier horizons, but Prottle scales better. In this problem mGPT can find the optimal solution faster than both Prottle and Paragraph, solving it in $\sim$0.1 seconds.

[7] http://rsise.anu.edu.au/~thiebaux/benchmarks/ppddl/

Horizon	PRTTL Time	NA-PG Time	Cost
10	14.0	0.23	0.728
15	21.6	0.73	0.607
20	25.1	12.5	0.486
25	36.0	52.2	0.429
30	40.6	103	0.429

(a) g-tire

Horizon	PRTTL Time	CA-PG Time	PRTTL Cost	CA-PG Cost
5	4.38	0.08	0.272	0.204
6	14.9	0.13	0.204	0.193
7	168	0.26	0.178	0.156
8	554	0.71	0.151	0.149
15	–	613	–	0.078

(b) maze

CCD-PG	CCI-PG	NCD-PG	NCI-PG
132	14.7	90.1	1.52

(c) machineshop

Planner	Horizon	Time	Cost
CA-PG	5	0.18	0.978
CA-PG	15	–	–
NA-PG	5	0.16	0.978
NA-PG	15	691	0.925
NCD-PG	∞	5.35	0.000
NCI-PG	∞	1.63	0.000
M-GPT	∞	15.8	0.000

(d) zeno-travel

Planner	Horizon	Time	Cost
PRTTL	3	2.29	0.344
PRTTL	5	74.5	0.344
CA-PG	3	0.11	0.344
CA-PG	5	0.56	0.344
NA-PG	3	0.07	1.000
NA-PG	5	0.09	0.344
CCD-PG	∞	0.72	0.344
CCI-PG	∞	1.57	0.344
NCD-PG	∞	0.14	0.344
NCI-PG	∞	0.21	0.344
M-GPT	∞	0.89	0.346

(e) teleport

Figure 5: Experimental Results.

The maze problem involves a number of connected rooms and doors, some of which are locked and require a specific key to open. This problem has some potential for concurrency, although mostly of the type not allowed in composite contingency plans. None of the planner configurations fully expoit it. Paragraph scales much better than Prottle this time.

In the machineshop problem, machines apply manufacturing tasks (such as paint, polish, etc.) to objects. This problem has a significant potential for concurrency. These results compare the performance of the cyclic planner configurations; the non-concurrent configurations do better than the concurrent, and iterative deepening does better than the depth-first search. This is because there exists a proper policy (with a zero cost) that the iterative deepening search can find without exploring the entire search space. The combination of acyclic and iterative deepening does better than mGPT, which solves this problem in 2.06 seconds.

In the zeno-travel problem, the goal to move a plane from one city to another without disturbing the people at the respective cities. Solutions to this problem are non-concurrent, and must repeat actions sequences until a low-probability outcome occurs, making this problem impractical for non-cyclic planners. The NCI-PG configuration solves this problem an order of magnitude faster than mGPT.

Finally, teleport is a problem where it is possible to move between 'linked' locations, and to change the pairs of locations that are linked. This domain admits concurrency, as demonstrated by the NA-PG configuration's inability to find a solution for horizon of 3. All of Paragraph's configurations are able to deal well with the branching in this domain in comparison to Prottle.

These results show that Paragraph is competitive with the state of the art in probabilistic planning. In general, Paragraph has the best comparative performance on problems with a high forward branching factor and relatively

few ways of achieving the goal.

Conclusion, Related, and Future Work

The contribution of this paper is an approach that extends the Graphplan framework for concurrent probabilistic planning. Optimal concurrent contingency plans are generated using state information to combine the trajectories that are found by Graphplan's goal-regression search. This requires storing an explicit trace of the backward search; a technique that has been investigated by Zimmerman and Kambhampati (2005) in the deterministic setting. The search retains the compression inherent in goal regression and benefits from mutex and nogood reasoning.

In contrast, Paragraph's predecessors PGraphplan and TGraphplan (Blum & Langford 1999) do not fully exploit the Graphplan framework. PGraphplan merely uses the planning graph as a reachability analysis tool and dispenses with Graphplan's techniques for efficiently managing concurrency. TGraphplan lacks the ability to explicitely reason about contigencies and their cost. Dearden *et. al* (2003) show how some of the shortcomings inherent in TGraphplan's approach can be remedied in the context of decision-theoretic temporal planning. They describe a method based on back-propagation through the planning graph to estimate the utility of individual contingencies. Prottle (Little, Aberdeen, & Thiébaux 2005) is more expressive than Paragraph, as it additionally reasons about (probabilistic) action durations and effect timings. But like PGraphplan, Prottle uses the planning graph primarily for computing heuristic estimates for a forward search.

Probabilistic planning approaches that are based on an explicit MDP formulations do not typically allow for concurrency; a notable exception is the work by Mausam and Weld (2004) on concurrent MDPs. The techniques they have developed for managing concurrency are orthogonal to ours. They include admissible pruning based on properties of the Bellman equation, and computing lower and upper bounds by solving weaker and stronger versions of the problem.

Whereas most of the current research on decision-theoretic and probabilistic planning is directed towards MDP models and algorithms, our work falls in the camp of some of the early attempts at extending classical (mostly partial order) planning to the probabilistic case, see (Onder & Pollack 1999) for a synthesis and references. Paragraph revisits this line of research; but by building on more recent advances in classical planning it is, unlike its predecessors, competitive with state of the art probabilistic planners. Given the recent spectacular progress in classical and temporal planning, we believe that this line of research is likely to florish again.

One of the main advantages of Paragraph's approach is that it can exploit much of the research into the Graphplan framework. We have already applied some of this research to Paragraph, but we would like to take this further. We have some evidence that a small amount of control knowledge in the form of mutex invariants can make a substantial impact on Paragraph's efficiency. This suggests that there would also be a benefit in investigating ways strengthening the planning graph's mutex reasoning. Currently, Paragraph relies on the optimisations built into the Graphplan framework, and is not able to additionally benefit from any search heuristics. We would like to investigate the possibilities for applying such heuristics to Paragraph. We also intend to extend Paragraph to the probabilistic temporal setting, where actions are durative in addition to being concurrent and having probabilistic effects.

Acknowledgements

We thank Rao Kambhampati and the reviewers for interesting discussions and comments, and National ICT Australia (NICTA) and the Australian Defence Science & Technology Organisation (DSTO) for their support, in particular via the DPOLP (Dynamic Planning, Optimisation & Learning) project. NICTA is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian National Research Council.

References

Aberdeen, D.; Thiébaux, S.; and Zhang, L. 2004. Decision-theoretic military operations planning. In *Proc. ICAPS*.

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

Blum, A., and Langford, J. 1999. Probabilistic planning in the Graphplan framework. In *Proc. ECP*.

Bonet, B., and Geffner, H. 2003. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. ICAPS*.

Bonet, B., and Geffner, H. 2005. mGPT: A probabilistic planner based on heuristic search. *Journal of Artificial Intelligence Research* 24:933–944.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research* 11:1–94.

Bresina, J.; Dearden, R.; Meuleau, N.; Ramakrishnan, S.; Smith, D.; and Washington, R. 2002. Planning under continuous time and resource uncertainty: A challenge for AI. In *Proc. UAI*.

Dearden, R.; Meuleau, N.; Ramakrishnan, S.; Smith, D.; and Washington, R. 2003. Incremental contingency planning. In *Proc. ICAPS Workshop on Planning under Uncertainty and Incomplete Information*.

Kambhampati, S.; Parker, E.; and Lambrecht, E. 1997. Understanding and extending graphplan. In *Proc. ECP*.

Koehler, J.; Nebel, B.; Hoffmann, J.; and Dimopoulos, Y. 1997. Extending planning graphs to an ADL subset. In *Proc. ECP*.

Little, I.; Aberdeen, D.; and Thiébaux, S. 2005. Prottle: A probabilistic temporal planner. In *Proc. AAAI*.

Long, D., and Fox, M. 1999. Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research* 10:87–115.

Mausam, and Weld, D. 2004. Solving concurrent Markov decision processes. In *Proc. AAAI*.

Mausam, and Weld, D. 2005. Concurrent probabilistic temporal planning. In *Proc. ICAPS*.

Onder, N., and Pollack, M. 1999. Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms. In *Proc. AAAI*.

Smith, D., and Weld, D. 1999. Temporal planning with mutual exclusion reasoning. In *Proc. IJCAI*.

Younes, H., and Littman, M. 2004. PPDDL1.0: The language for the probabilistic part of IPC-4. In *Proc. International Planning Competition*.

Zimmerman, T., and Kambhampati, S. 2005. Using memory to transform search on the planning graph. *Journal of Artificial Intelligence Research* 23:533–585.